Algebra-based identification of tree patterns in XQuery

Andrei Arion[1,2], Véronique Benzaken[2], Ioana Manolescu[1], Yannis Papakonstantinou[3], and Ravi Vijay[1,4]

[1] INRIA Futurs, Gemo group, France   `firstname.lastname@inria.fr`
[2] LRI, Univ. Paris 11, France   `veronique.benzaken@lri.fr`
[3] CSE Dept., UCSD, USA   `yannis@cs.ucsd.edu`
[4] IIT Bombay, India   `ravivj@cse.iitb.ac.in`

Abstract. Query processing performance in XML databases can be greatly enhanced by the usage of materialized views whose content has been stored in the database. This requires a method for identifying query subexpressions matching the views, a process known as view-based query rewriting. This process is quite complex for relational databases, and all the more daunting on XML databases.

Current XML materialized view proposals are based on tree patterns, since query navigation is conceptually close to such patterns. However, the existing algorithms for extracting tree patterns from XQuery do not detect patterns *across nested query blocks*. Thus, complex, useful tree pattern views may be missed by the rewriting algorithm. We present a novel tree pattern extraction algorithm from XQuery queries, able to identify larger patterns than previous methods. Our algorithm has been implemented in an XML database prototype [5].

1   Introduction

The XQuery language [23] is currently gaining adoption as the standard query language for XML. One performance-enhancing technique in XQuery processing is the usage of materialized views. The idea is to pre-compute and store in the database the result of some queries (commonly called *view definitions*), and when a user query arrives, to identify which parts of the query match one of the pre-computed views. The larger parts of the query one can match with a view, the more efficient query processing will be, since a bigger part of the query computation can be obtained directly from the materialized view.

Identifying useful views for a query requires reasoning about containment (e.g., is all the data in view v contained in the result of query q ?) and equivalence (e.g., is the join of views v_1 and v_2 equivalent to the query q ?). XML query ontainment and equivalence are well understood when views and queries are represented as *tree patterns*, containing tuples of elements satisfying specific structural relationships [18, 19]. Moreover, popular XML indexing and fragmentation strategies also materialize tree patterns [10, 11, 13, 14]. Therefore, tree patterns are an interesting model for XML materialized views [3, 5, 6, 11, 12].

Our work is placed in the context of XQuery processing based on a persistent store. We make some simple assumptions on this context, briefly presented next.

Most persistent XML stores assign some *persistent identifiers* to XML elements. Such identifiers are often *structural*, that is, by comparing the identifiers id_1 and id_2 of two elements e_1 and e_2, one can decide whether some structural relationship exists between e_1 and e_2: for instance, whether e_1 is a child, parent, or sibling of e_2. The interest of structural identifiers is that establishing such relationships directly is much more efficient than navigating from e_1 to e_2 in the database to verify it. Numerous structural ID proposals have been made so far, see e.g. [2, 20]. *We assume persistent IDs are available in the store*. The IDs may, but do not need to, have structural properties.

Our second assumption is that a materialized view may store: (*i*) *node IDs* [10, 12, 14], (*ii*) node *values* (i.e., the text nodes directly under an element, or the value of an attribute) [10], and/or (*iii*) node *content*, that is, the full subtree rooted at an XML element (or a pointer to that subtree) [6]. This assumption provides for flexible view granularity.

To take advantage of tree pattern-shaped materialized views, one has to understand which views can be used for a query q. This process can be seen as a translating q to some *query patterns* $p_{q1}, \ldots, p_{qn}$, followed by a rewriting of every query pattern p_{qi} using the view patterns $p_{v1}, \ldots, p_{vm}$. The first step (query-to-pattern translation) is crucial. Intuitively, the bigger the query patterns, the bigger the view(s) that can be used to rewrite them, thus the less computations remain to be applied on top of the views.
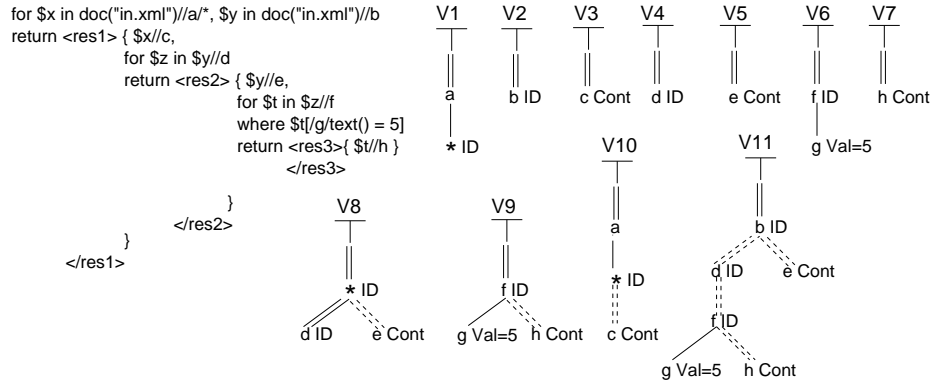
The contribution of this paper is a provably correct algorithm identifying tree patterns in queries expressed in a large XQuery subset. The advantage of this method over existing ones [6, 9, 21] is that the patterns we identify are strictly larger than in previous works, and in particular may span over nested XQuery blocks, which was not the case in previous approaches. We ground our algorithm on an algebra, since (as we will show) the translation is quite complex due to XQuery complexity, and straightforward translation methods may loose the subtle semantic relationships between a pattern and a query.

Materialized views: advantages and drawbacks A legitimate question is whether the cost of materializing and maintaining materialized views is justified by the advantages they provide ? It turns out that in XML persistent store, a tree-based approach is rarely (if ever !) sufficient to support complex querying. We survey XML storage and indexing strategies in [15]. Shredding schemes (aiming at loading XML documents in a set of relational tables) also offer an example of materialized XML views, recognized as such in [11, 16]. XML view maintenance in the presence of updates is a direction we are currently working on.

The paper is organized as follows. Section 2 motivates the need for pattern recognition in XQuery queries. Section 3 sets the formal background for the translation algorithm presented in Section 4.

2 Motivating example

We illustrate the benefits of our tree pattern extraction approach on the sample XQuery query in Fig. 1, featuring three nested for-where-return blocks. An important XQuery feature to keep in mind is that when a return clause constructs new elements, if an expression found in the element constructor evaluates to $\emptyset$ (the empty result), an element must still be constructed, albeit with no content corresponding to that particular expression. For instance,

```
for $x in doc("in.xml")//a/*, $y in doc("in.xml")//b
return <res1> { $x//c,
            for $z in $y//d
            return <res2> { $y//e,
                        for $t in $z//f
                        where $t[/g/text() = 5]
                        return <res3>{ $t//h }
                               </res3>
                    }
                </res2>
            }
        </res1>
```

V1: a, * ID

V2: b ID

V3: c Cont

V4: d ID

V5: e Cont

V6: f ID, g Val=5

V7: h Cont

V8: * ID, d ID, e Cont

V9: f ID, g Val=5, h Cont

V10: a, * ID, c Cont

V11: b ID, d ID, e Cont, f ID, g Val=5, h Cont

Fig. 1. Sample XQuery query and corresponding tree patterns/views.

in Fig. 1, if for some bindings of the variables $x and $y, the expression $x//c yields an empty result, a res1 element will still be constructed, with no content corresponding to $x//c (but perhaps with some content produced by the nested for-where-return expression).

Next to the query, Fig. 1 depicts eleven possible corresponding query tree patterns. Each pattern is rooted at the ⊤ symbol, denoting the document root. Pattern edges may be labeled / for parent-child relationships, or // for ancestor-descendent relationships. Pattern nodes may be labeled with node names or with ∗ (any name). When a pattern node carries an *ID* symbol, the pattern is said to *contain the ID* of the XML nodes corresponding to the pattern node; similarly, if a pattern node is labeled *Cont* (respectively, *Val*), the pattern is said to *contain the contents* (respectively, the value) of XML nodes corresponding to the pattern node. If a pattern node is annotated with a *Val = c* predicate, for some constant *c*, then only XML nodes whose value satisfies that predicate (and the structural constraints on the node) will belong to the pattern.

We still need to explain the meaning of dashed pattern edges. These edges are *optional* in the following sense: an XML node matching the upper (parent/ancestor) node of a dashed edge may lack XML descendents matching the lower (child/descendent) node, yet that node may still belong to the pattern (if the edge was not optional, this would not be the case). If the lower node of a dashed edge was annotated with *ID*, *Val* or *Cont*, the pattern will contain some null (⊥) values to account for the missing children/descendents.

As previously mentioned, patterns play a dual role in our approach: view definitions, and query sub-expressions. Thus, each pattern $V_1, \ldots, V_{11}$ is a subexpression of the query at left, and (for our explanation) we also assume it is available as a materialized view. When a pattern is interpreted as a view, we say it stores various *ID*, *Cont* and *Val* attributes; when it is interpreted as a query subexpression, we say it needs such attributes.

Let us now compare the ability of different algorithms to recognize the patterns in the query (thus, enable their usage for view-based query rewriting).

Several existing view-based XML query rewriting frameworks [6, 24] concentrate on XPath views, storing data for one pattern node only (since XPath queries have one return node), and lacking optional edges. Similar indexes are described in [10, 14]. In Fig. 1, the only XPath views are V_1-V_7, which represent the largest XPath patterns that one can derive from the query in Fig. 1; they store *Cont* for all nodes which must be returned (such as the c, e and h nodes), and *ID* for all nodes whose values or content are not needed for

the query, but which must be traversed by the query (such as the $a/*$, b, d nodes etc.) In this case, the only way to answer the query is to perform five joins and a cartesian product (the latter due to the fact that x and y are not connected in any way) to connect the data from V_1-V_7. This approach has some disadvantages. First, it needs an important amount of computations, and second, it may lead to reading from disk more data than needed (for instance, V_7 contains all h elements, while the query only needs those h elements under $//b//d//f$).

The algorithms of [9, 21] extract patterns storing information from several nodes, and having optional edges. However, these patterns are not allowed to span across nested for-where-return expressions. In Fig. 1, this approach would extract the patterns V_2, V_{10}, V_8 and V_9, thus the query can be rewritten by joining the corresponding views. This still requires three joins, and may lead to read data from many elements not useful to the query.

Our algorithm extracts from the query in Fig. 1 only two patterns: V_{10} and V_{11}. Based on these, we rewrite the query by a single join (more exactly, a cartesian product) of the corresponding V_{10} and V_{11} views, likely to be much less expensive than the other approaches.

Is a formal model required to describe pattern extraction ? The answer is yes, because one needs to model precisely (i) query semantics, typically using an algebra [7, 17, 22] and (ii) view semantics; in [4] we provided the full algebraic semantic of patterns such as those in Fig. 1. A formal model is needed, to ensure the patterns have exactly the same meaning as query subexpressions, or, when this is not the case, to compute *compensating actions* on the views. For instance, consider V_{11} in Fig. 1. Here, the d and the e nodes are optional descendents of the b nodes, and so they should be, according to the query. However, due to the query nesting, no e element should appear in the result, if its b ancestor does not have d descendents. This $d \rightarrow e$ dependency is not expressed by V_{11}, and is not expressible by any tree pattern, because such patterns only account for ancestor-descendent relationships. Thus, V_{11} is the best possible tree pattern view for the part of the query related to variable y, yet it is not exactly what we need. An (inexpensive) selection on V_{11}, on the condition $(d.ID \neq \perp) \vee (d.ID = \perp \wedge e.Cont = \perp)$, needs to be applied to adapt the view to the query.

For simplicity, in this section, no nesting or grouping has been considered, neither in the patterns in Fig. 1, nor in the query rewriting strategies. However, given that XQuery does construct complex grouped results (e.g., all c descendents of the same x must be output together in Fig. 1), pattern models considered in [4, 21], as well as our translation method, do take nesting into account.

3 Data model, algebra, and query language

3.1 Data model

Let $\mathcal{A}$ be an infinite alphabet and $\mathcal{L}$, $\mathcal{I}$ be two disjoint subsets of $\mathcal{A}$. A special $\mathcal{A}$ constant ϵ denotes the empty string. We view an XML document as an unranked labeled ordered tree. Any node has a tag, corresponding to the element or attribute name, and may have a value. Attribute and element names range over $\mathcal{L}$, while values range over $\mathcal{A}$. The *value* of a node n belongs to $\mathcal{A}$ and is obtained by concatenating the text content of all children of n in document order; the result may be ϵ if the node does not have text children. The *content* of a node n is an $\mathcal{A}$ value, obtained by serializing the labels and values of all nodes from the tree rooted in n, in a top-down, left-to-right traversal. Nodes have unique *identities*.
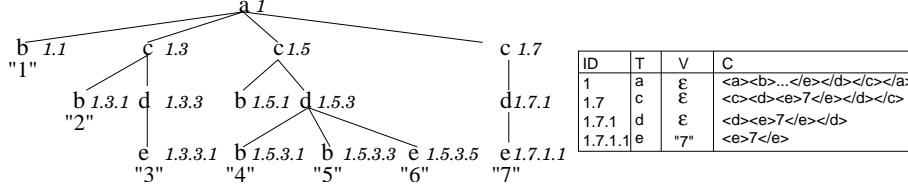
ID	T	V	C
1	a	ε	<a><b>...</e></d></c>
1.7	c	ε	<c><d><e>7</e></d></c>
1.7.1	d	ε	<d><e>7</e></d>
1.7.1.1	e	"7"	<e>7</e>

Fig. 2. Sample XML document and some tuples from its canonical relation.

Let n_1, n_2 be two XML nodes. We denote the fact that n_1 is n_2's parent as $n_1 \prec n_2$, and the fact that n_1 is an ancestor of n_2 as $n_1 \lll n_2$. We extend this notation to element IDs; $i_1 \prec i_2$ (resp. $i_1 \lll i_2$) *iff* i_1 identifies n_1, i_2 identifies n_2 and $n_1 \prec n_2$ (resp. $n_1 \lll n_2$).

We assume available an ID scheme I, that is, an injective function assigning to every node a value in $\mathcal{I}$. Figure 2 shows a simple XML document, where nodes are given structural ORDPATH identifiers [20].

We will rely on a nested relational model [1] as follows. The value of a tuple attribute is either a value from $\mathcal{A}$, or null ($\perp$), or a collection (set, list or bag) of homogeneous tuples. Notice the alternation between the tuple and the collection constructors. We use lowercase letters for relation names, and uppercase letters for attribute names, as in $r(A_1, A_2(A_{21}, A_{22}))$. Values are designated by lowercase letters. For instance, a tuple in $r(A_1, A_2(A_{21}, A_{22}))$ may have the value $t(x_1, [(x_3, \perp)\ (x_4, x_5)])$.

The basic ingredient of the algebraic expressions used in our translation method is a (virtual) relation capturing the data associated to an XML element. Given a document d, the *canonical element relation* $e_d(ID, T, V, C) \subseteq \mathcal{I} \times \mathcal{L} \times \mathcal{A} \times \mathcal{A}$ contains, for every element $n \in d$, a 4-tuple consisting of: the ID assigned to n by I; n's tag; n's value; and n's content. A canonical attribute relation can be similarly defined. Without loss of generality, we will only refer to e_d. For example, Figure 2 shows some tuples from the canonical element relation corresponding to the sample XML document. For simplicity, from now on, we will omit the d index and refer to the canonical relation simply as e. Furthermore, we will use e_x, where x is some element name, as a shorthand for $\sigma_{T=x}(e)$.

3.2 Logical algebra

To every nested relation r, corresponds a Scan operator, also denoted r, returning the (possibly nested) corresponding tuples. Other standard operators are the cartesian product $\times$, the union $\cup$ and the set difference $\setminus$ (which do not eliminate duplicates).

We consider predicates of the form $A_i\ \theta\ c$ or $A_i\ \theta\ A_j$, where c is a constant. θ ranges over the comparators $\{=, \leq, \geq, <, >, \prec, \lll\}$, and $\prec$, $\lll$ only apply to $\mathcal{I}$ values.

Let *pred* be a predicate over atomic attributes from r, or r and s. Selections σ_{pred} have the usual semantics. A join $r \bowtie_{pred} s$ is defined as $\sigma_{pred}(r \times s)$. For convenience, we will also use outerjoins $\bowtie\!\!\!\!\!\_\,_{pred}$ and semijoins $\ltimes_{pred}$ (although strictly speaking they are redundant to the algebra). Another set of redundant, yet useful operators, are *nested joins*, denoted $\bowtie^n_{pred}$, and *nested outerjoins*, denoted $\bowtie\!\!\!\!\!\_\,^n_{pred}$, with the following semantics:

$$r \bowtie\!\!\!\!\!\_\,^n_{pred} s = \{(t_1, \{t_2 \in s \mid pred(t_1, t_2)\}) \mid t_1 \in r\}$$
$$r \bowtie^n_{pred} s = \{(t_1, \{t_2 \in s \mid pred(t_1, t_2)\})\ )\mid t_1 \in r, \{t_2 \in s \mid pred(t_1, t_2)\} \neq \emptyset\ \}$$

An interesting class of logical join operators (resp. nested joins, outerjoins, nested outer-joins, or semijoins) is obtained when the predicate's comparator is $\prec$ or $\prec\!\!\prec$, and the operand attributes are identifiers from $\mathcal{I}$. Such operators are called *structural joins*. Observe that we only refer to *logical structural joins*, independently of any physical implementation algorithm; different algorithms can be devised [2, 8].

Let $A_1, A_2, \ldots, A_k$ be some atomic r attributes. A projection $\pi_{A1,A2,\ldots,Ak}(r)$ by default does not eliminate duplicates. Duplicate-eliminating projections are singled out by a superscript, as in π^0. The group-by operator $\gamma_{A_1,A_2,\ldots,A_k}$, and unnest u_B, where B is a collection attribute, have the usual semantics [1].

We use the *map* meta-operator to define algebraic operators which apply *inside* nested tuples. Let op be a unary operator, $r.A_1.A_2.\ldots.A_{k-1}$ a collection attribute, and $r.A_1.A_2.\ldots.A_k$ an atomic attribute. Then, $map(op, r, A_1.A_2.\ldots.A_k)$ is a unary operator, and:

- If $k = 1$, $map(op, r, A_1.A_2.\ldots.A_k) = op(r)$.
- If $k > 1$, for every tuple $t \in r$:
  - If for every collection $r' \in t.A_1$, $map(op, r', A_2.\ldots.A_k) = \emptyset$, t is eliminated.
  - Otherwise, a tuple t' is returned, obtained from t by replacing every collection $r' \in r.A_1$ with $map(op, r', A_2.\ldots.A_k)$.

For instance, let $r(A_1(A_{11}, A_{12}), A_2)$ be a nested relation. Then, $map(\sigma_{=5}, r, A_1.A_{11})$ only returns those r tuples t for which *some* value in $t.A_1.A_{11}$ is 5 (existential semantics), and reduces these tuples accordingly. *Map* applies similarly to π, γ and u. By a slight abuse of notation, we will refer to $map(op, r, A_1.A_2.\ldots.A_k)$ as $op_{A_1.A_2.\ldots.A_k}(r)$. For instance, the sample selection above will be denoted $\sigma_{A_1.A_{11}=5}(r)$.

Binary operators are similarly extended, via *map*, to nested tuples (details omitted).

The xml_{templ} operator wraps an input tuple into a single piece of XML text, by gluing together (some of) its attributes, and possibly adding tags, as specified by the tagging template $templ$. For every tuple t, whose data has already been grouped and structured, xml_{templ} thus outputs an $\mathcal{A}$ value which is the content of the newly created element. While this is slightly different from new node construction (as xml_{templ} does not create a new node identity), we use it here for simplicity and without loss of generality. Element construction operators closer to XQuery semantics [17, 22] could also be used.

3.3   Query language

We consider a subset of XQuery, denoted $\mathcal{Q}$, obtained as follows.

(1) XPath$^{\{/,//,*,[\,]\}} \subset \mathcal{Q}$, that is, any core XPath [18] query over some document d is in $\mathcal{Q}$. We allow in such expressions the usage of the function *text()*, which on our data model returns the value of the node it is applied on. This represents a subset of XPath's absolute path expressions, whose navigation starts from the document root. Examples include $/a/b$ or $//c[//d/text() = 5]/e$. Navigation branches enclosed in [ ] may include complex paths and comparisons between a node and a constant $c \in \mathcal{A}$. Predicates connecting two nodes are not allowed; they may be expressed in XQuery for-where syntax (see below). **(2)** Let $\$x$ be a variable bound in the query context [23] to a list of XML nodes, and p be a core XPath expression. Then, $\$x\,p$ belongs to $\mathcal{Q}$, and represents the path expression p applied with $\$x$'s bindings list as initial context list. For instance, $\$x/a[c]$ returns the a children of

x bindings having a c child, while $x//b$ returns the b descendents of x bindings. This class captures *relative* XPath expressions in the case where the context list is obtained from some variable bindings. We denote the set of expressions (1) and (2) above as $\mathcal{P}$, the set of path expressions. **(3)** For any two expressions e_1 and $e_2 \in \mathcal{Q}$, their concatenation, denoted e_1, e_2, also belongs to $\mathcal{Q}$. **(4)** If $t \in \mathcal{L}$ and $exp \in \mathcal{Q}$, element constructors of the form $\langle t \rangle \{exp\} \langle /t \rangle$ belong to $\mathcal{Q}$. **(5)** All expressions of the following form belong to $\mathcal{Q}$:

$$\boxed{xq} \quad \begin{aligned} &\text{for } \$x_1 \text{ in } p_1,\ \$x_2 \text{ in } p_2,\ \ldots,\ \$x_k \text{ in } p_k \\ &\text{where } p_{k+1}\ \theta_1\ p_{k+2} \text{ and } \ldots \text{ and } p_{m-1}\ \theta_l\ p_m \\ &\text{return } q(x_1, x_2, \ldots, x_k) \end{aligned}$$

where $p_1, p_2, \ldots, p_k, p_{k+1}, \ldots, p_m \in \mathcal{P}$, any p_i starts either from the root of some document d, or from a variable x_l introduced in the query before p_i, $\theta_1, \ldots, \theta_l$ are some comparators, and $q(x_1, \ldots, x_k) \in \mathcal{Q}$. Note that the return clause of a query may contain several other for-where-return queries, nested and/or concatenated and/or grouped inside constructed elements. The query in Figure 1 illustrates our supported fragment.

4 Pattern extraction algorithms

Our algorithm proceeds in two steps. First, $\mathcal{Q}$ queries are translated into expressions in the algebra previously described; Sections 4.1 and 4.2 explain this translation. Second, algebraic equivalence and decomposition rules are applied to identify, in the resulting expressions, subexpressions corresponding to tree patterns. The algebraic rules are quite straightforward. The ability to recognize pattern subexpressions is due to the formal algebraic pattern semantics provided in a previous work [4]. Section 4.3 illustrates it on our running example.

Queries are translated to algebraic expressions producing one $\mathcal{A}$ attribute, corresponding to the serialized query result. We describe query translation as a translation function $alg(q)$ for every $q \in \mathcal{Q}$. We will also use an auxiliary function $full$; intuitively, $full$ returns "larger" algebraic expressions, out of which alg is easily computed.

4.1 Algebraic translation of path queries

For any $q \in \mathcal{P}$, let $ret(q)$ denote the return node of q. Let d be a document, and a an element name. Then:

$$full(d//*) \stackrel{\text{def}}{=} e, \text{ and } alg(d//*) \stackrel{\text{def}}{=} \pi_C(e)$$
$$full(d//a) \stackrel{\text{def}}{=} (e_a), \text{ and } alg(d//a) \stackrel{\text{def}}{=} \pi_C(e_a)$$

Translating $d/*$ and d/a requires care to separate just the root element from e:

$$full(d/*) \stackrel{\text{def}}{=} e_1 \setminus \pi_{e_3}(e_2 \bowtie_{e_2.ID \prec e_3.ID} e_3), \text{ and } alg(d/*) \stackrel{\text{def}}{=} \pi_C(full(d/*))$$

where e_1, e_2 and e_3 are three occurences of the e relation, $e_2.ID$ (respectively, $e_3.ID$) is the ID attribute in e_2 (respectively e_3), and the projection π_{e_3} retains only the attributes of e_3. The set difference computes the e tuple corresponding to the element that does not have a parent in e (thus, the root element). Similarly,

$$full(d/a) \stackrel{\text{def}}{=} e_a \setminus \pi_{e_3}(e_2 \bowtie_{e_2.ID \prec e_3.ID} e_3), \text{ and } alg(d/a) \stackrel{\text{def}}{=} \pi_C(full(d/a))$$

In general, for any $\mathcal{P}$ query q:

- If q ends in $text()$, then $alg(q) = \pi_{V_{last}}(\pi^0(full(q)))$, where V_{last} is the V attribute from the e_d relation corresponding to $ret(q)$. The inner projection π^0 eliminates possible duplicate nodes, in accordance with XPath semantics [23]. The outer projection ensures only the text value is retained.
- If q does not end in $text()$, then $alg(q) = \pi_{C_{last}}(\pi^0(full(q)))$, where C_{last} is the C attribute from the e_d relation corresponding to $ret(q)$.

Note that the resulting algebraic expressions return node *value* or *content*, while in general XPath queries may return *nodes*. Alternatively, node identifiers can be returned by setting, for node-selecting XPath queries, $alg(q) \stackrel{\text{def}}{=} \pi_{ID_{last}}(\pi^0(full(q)))$, where ID_{last} is the ID attribute from the e_d relation corresponding to $ret(q)$. Since XPath results frequently need to be returned in a serialized form, e.g., to be shown to a user, or sent in a Web service, we consider the C attribute is really returned, thus use $\pi_{C_{last}}$ in the translation.

We now focus on defining the $full$ algebraic function for path queries, keeping in mind how alg derives from $full$ for such queries. For any query $q \in \mathcal{P}$, we have:

$$full(q//a) \stackrel{\text{def}}{=} full(q) \bowtie_{e_q.ID \ll e_a.ID} e_a$$

where $e_q.ID$ is the ID attribute in $full(q)$ corresponding to $ret(q)$, while $e_a.ID$ is the ID from the e_a relation at right in the above formula. When $//$ is replaced with $/$, the translation involves $\prec$ instead of $\ll$. We also have:

$$full(q[text() = c]) \stackrel{\text{def}}{=} \sigma_{V=c}(full(q))$$

If $q_1 \in \mathcal{P}$ and $q_2 \in \text{XPath}^{\{/,//,*,[]\}}$ is a relative path expression starting with a child navigation step, we have:

$$full(q_1[q_2]) \stackrel{\text{def}}{=} full(q_1) \ltimes_{e_1.ID \prec e_2.ID} full(//q_2)$$

where $e_1.ID$ is the ID corresponding to $ret(q_1)$, $//q_2$ is an absolute path expression obtained by adding a descendent navigation step, starting from the root, in front of q_2, and $e_2.ID$ corresponds to the first node of q_2. Here and from now on, we consider all relative path expressions start with a child step. If the first step is to a descendent, $\prec$ should be replaced with $\ll$ in the translation.

Let $\$x$ be a variable bound to the result of query $q_{\$x}$, and q be a relative path expression starting with a child navigation step. Then:

$$full(\$x\ q) \stackrel{\text{def}}{=} full(q_{\$x}) \bowtie_{e_1.ID \prec e_2.ID} (full(q))$$

where $e_1.ID$ is the ID corresponding to $ret(q_{\$x})$, and $e_2.ID$ is the ID corresponding to the top node in $full(q)$.

Example. Consider the path expressions $p_{\$x} = //a/*$, $p_{\$y} = //b$, $p_{\$z} = \$y//d$ and $p_{\$t} = \$z//f$ (see Fig. 1). Applying the above rules, we obtain:

$$full(p_{\$x}) = e_a \bowtie_{e_a.ID \prec e.ID} e, \quad full(p_{\$y}) = e_b$$
$$full(p_{\$z}) = full(p_{\$y}) \bowtie_{e_{\$y}.ID \ll e_d.ID} e_d,$$
$$full(p_{\$t}) = full(p_{\$z}) \bowtie_{e_{\$z}.ID \ll e_f.ID} e_f$$

Now consider the path expressions $p_1 = \$x//c$, $p_2 = \$y//e$, $p_3 = \$t[g/text() = 5]$ and $p_4 = \$t//h$, also extracted from the query in Fig. 1. We have:

$$full(p_1) = full(p_{\$x}) \bowtie_{e_{\$x}.ID \preceq\!\!\!< e_c.ID} e_c, \ full(p_2) = full(p_{\$y}) \bowtie_{e_{\$y}.ID \preceq\!\!\!< e_e.ID} e_e,$$
$$full(p_3) = full(p_{\$t}) \bowtie_{e_{\$t}.ID \prec e_g.ID} \sigma_{V=5}(e_g),$$
$$full(p_4) = full(p_{\$t}) \bowtie_{e_{\$t}.ID \preceq\!\!\!< e_h.ID} e_h$$

In the above, $e_{\$y}$, $e_{\$z}$ and $e_{\$t}$ are the e relations corresponding to the return nodes in the translations of $p_{\$x}$, $p_{\$y}$ and $p_{\$t}$. The alg expressions are easily obtained from $full$.

> for $\$x_1$ in p_1, $\$x_2$ in $\$x_1/p_2,\ldots,\$x_k$ in $\$x_1/p_k$
> $\boxed{xq_1}$ where $\$x_1/p_{k+1} \ \theta \ \$x_1/p_{k+2}$ and $\ldots \$x_1/p_{m-1} \ \theta \ \$x_1/p_m$
> return $\$x_1/p_{m+1}, \$x_1/p_{m+2}, \ldots, \$x_1/p_n$

$full(xq_1) \overset{\text{def}}{=}$
$\sigma_{A_{k+1} \ \theta \ A_{k+2}, \ldots, A_{m-1} \ \theta \ A_m} (full(p_1) \bowtie_{ID_1 \prec ID_2} full(//p_2) \ldots \bowtie_{ID_1 \prec ID_k} full(//p_k)$
$\quad \bowtie^n_{ID_1 \prec ID_{k+1}} full(//p_{k+1}) \bowtie^n_{ID_1 \prec ID_{k+2}} \cdots \bowtie^n_{ID_1 \prec ID_m} full(//p_m)$
$\quad \bowtie^n_{ID_1 \prec ID_{m+1}} full(//p_{m+1}) \bowtie^n_{ID_1 \prec ID_{m+2}} \cdots \bowtie^n_{ID_1 \prec ID_n} full(//p_n) \ )$

$alg(xq_1) \overset{\text{def}}{=} \pi_{A_{m+1}, A_{m+2}, \ldots, A_n} (full(xq_1))$

Fig. 3. Generic XQuery query with simple return expression.

4.2 Algebraic translation of more complex queries

This section describes the translation of $\mathcal{Q}$ queries other than path expressions.

Concatenation We have $alg(q_1, q_2) \overset{\text{def}}{=} alg(q_1) \parallel alg(q_2)$ and $full(q_1, q_2) \overset{\text{def}}{=} full(q_1) \parallel full(q_2)$, where , denotes query concatenation, and $\parallel$ concatenation of tuple lists.

Element constructors Element constructor queries are translated by the following rule:

$$alg(\langle t \rangle \{q\} \langle /t \rangle) \overset{\text{def}}{=} xml(n(alg(q)), \langle t \rangle A_1 \langle /t \rangle)$$

where the nest operator n packs all tuples from $alg(q)$ in a single tuple with a single collection attribute named A_1. The second argument of the xml operator is a tagging template, indicating that values of the attribute named A_1 have to be packed in t elements. Furthermore, $full(\langle t \rangle \{q\} \langle /t \rangle) = n(full(q))$.

For-where-return expressions The translation rules for such query expressions are outlined in Fig. 3 and Fig. 4. For simplicity, these rules use a single θ symbol for some arbitrary, potentially different, comparison operators.

(1) Simple return clauses In the generic query xq_1 (Fig. 3), path expression p_1 is absolute, while all others are relative and start from a query variable $\$x_1$. Attribute ID_1 corresponds to $ret(p_1)$. The query returns some variables. Attribute ID_i is the attribute in $full(//p_i)$ corresponding to the top node of p_i, for every path expression p_i in $p_2, \ldots, p_m$. Attributes A_{k+1}, A_{k+2}, $\ldots$, A_m are those returned by the algebraic translations of the relative path expressions of the where clause, more precisely, the attributes in $alg(//p_{k+1})$, $alg(//p_{k+2})$, $\ldots$, $alg(//p_m)$. Each A_{k+i} is V or C, depending on p_{k+i}. Note that once $//$ is added in front of such a relative path expression, $//p_{k+i}$ is an absolute expression, thus translatable to the algebra. The child navigation step connecting $\$x_1$ and an expression p_{k+i}

$$\boxed{xq_2}\quad \begin{array}{l} \text{for } \$x \text{ in } p_f \text{ where } pred(p_w(\$x)) \\ \text{return } fwr(\$x) \end{array}$$

$$full(xq_2) \stackrel{\text{def}}{=} \sigma_{pred}(full(p_f) \bowtie^n_{ID_1 \prec ID_2} full(//p_w) \bowtie_{ID_1 = ID_1} full(fwr(p_f)))$$
$$alg(xq_2) \stackrel{\text{def}}{=} \pi_{fwr}(full(xq_2)), \text{ respectively } xml_{templ(fwr)}(\pi_{fwr}(full(xq_2)))$$

$$\boxed{xq_3}\quad \begin{array}{l} \text{for } \$x \text{ in } p_f \text{ where } pred(p_w(\$x)) \\ \text{return } \langle a \rangle \ \{ \ fwr(\$x) \ \} \ \langle /a \rangle \end{array}$$

$$full(xq_3) \stackrel{\text{def}}{=} \sigma_{pred}(full(p_f) \bowtie^n_{ID_1 \prec ID_2} full(//p_w) \bowtie\hspace{-0.9em}\sqsubset^n_{ID_1 = ID_1} full(fwr(p_f)))$$
$$alg(xq_3) \stackrel{\text{def}}{=} xml_{\langle a \rangle \cdot \langle /a \rangle}(\pi_{fwr}(full(xq_3))), \text{ resp. } xml_{\langle a \rangle templ(fwr) \langle /a \rangle}(\pi_{fwr}(full(xq_3)))$$

Fig. 4. Generic XQuery queries with complex return clauses.

is captured by the join $\bowtie^n_{ID_1 \prec ID_i}$. As an effect of this nested structural join, A_i may be nested in σ's input, therefore, the selection has existential semantics (recall the map-based extension of σ to nested attributes from Section 3.2).

The xq_1 rule easily extends to queries where the for clause features several unrelated variables, the where clause contains predicates over one or two variables, and the return clause returns only variables. Each subquery corresponding to an independent variable in the for clause is then translated separatedly, and the resulting expressions are joined. From now on, without loss of generality, we will use one variable in each for clause; adding more variables depending on the first one leads to structural join subexpression in the style of $full(xq_1)$, while adding more unrelated variables leads to value joins as sketched above.

(2) Nested for-where-return queries Such queries are illustrated by xq_2 and xq_3 in Fig. 4. Here, p_f is an absolute path expression, p_w a relative one, $pred$ a simple comparison predicate, and fwr a (potentially complex, nested) for-where-return query.

(2.1) The outer query does not construct new elements This is the case for xq_2 in Fig. 4. In $full(xq_2)$, ID_1 corresponds to $ret(p_f)$ and ID_2 to the top node in p_w. We add $//$ in front of p_w to make it absolute. The query $fwr(p_f)$ is obtained from fwr by adding a new "for" variable $\$x'$ bound to p_f, and replacing $\$x$ by $\$x'$. Thus, $fwr(p_f)$ is decorrelated from (it does no longer depend on) $\$x$; the dependency is replaced by the join on ID_1. In $alg(xq_2)$, the projection π_{fwr} retains only the attributes from $alg(fwr(p_f))$. Two alternatives exist for $alg(xq_2)$, as shown in Fig. 4:

- If fwr does not construct new elements, the query (and its translation) recall xq_1.
- If fwr constructs new elements, the top operator in $alg(fwr(p_f))$ is $xml_{templ(fwr)}$, for some given tagging template $templ(fwr)$. In this case, $full(xq_2)$ is built using exactly the same template. Note how the XML constructor "sifts up" as the top algebraic operator in the translation, in this case, from $alg(fwr)$ to $alg(xq_2)$. All algebraic translations have at most one xml operator.

(2.2) The outer query constructs new elements Query xq_3 in Fig. 4 encloses the results of some correlated query $fwr(\$x)$ in $\langle a \rangle$ elements. Therefore, in $full(xq_3)$ an outerjoin is used to ensure that xq_3 produces some output even for $\$x$ bindings for which $fwr(\$x)$ has an empty result. The outerjoin is nested, because all the results of $fwr(\$x)$ generated for a given $\$x$ must be included in a single $\langle a \rangle$ element. For $alg(xq_3)$, there are

again two cases. If fwr does not construct new elements, and since xq_3 does, the tagging template is a simple $\langle a \rangle$ element. If fwr also constructs some elements, the xml operator in $alg(xq_3)$ builds a bigger tagging template, by enclosing $templ(fwr)$ in an $\langle a \rangle$ element.

The translation of more complex $\mathcal{Q}$ queries can be derived from the above rules.

Example. Let us translate the query q in Fig. 1 (also recall the path expressions at the end of Section 4.1, and their translations). We can write q as:

> for \$x in $p_{\$x}$, \$y in $p_{\$y}$
> return $\langle res1 \rangle$\{ p_1,
> $\qquad\qquad\qquad \langle res2 \rangle$\{ p_2, for \$z in $p_{\$z}$ where p_3 return $\langle res3 \rangle$\{ p_4 \}$\langle /res3 \rangle \langle /res2 \rangle$ \}
> $\qquad\quad \langle /res1 \rangle$

which can be furthermore abstracted into:

> for \$x in $p_{\$x}$, \$y in $p_{\$y}$ return $\langle res1 \rangle$\{ p_1, $\langle res2 \rangle$\{ p_2, q_2 \}$\langle /res2 \rangle$ \} $\langle /res1 \rangle$

where query q_2 is: for \$z in $p_{\$z}$ where p_3 return $\langle res3 \rangle$\{ p_4 \}$\langle /res3 \rangle$. Let us first translate q_2. Applying the xq_3 translation rule from Fig. 4, we obtain:

$$ full(q_2) = full(p_{\$z}) \bowtie^n_{e_{\$z}.ID \prec e_3.ID} full(//p_3) \ltimes^n_{e_{\$z}.ID = e_{\$z}.ID} full(p_{\$z}//p_4) $$

Applying the xq_3 rule again, twice, for q leads to:

$$ (*)\ full(q) = full(p_{\$x}) \times full(p_{\$y}) \ltimes^n_{e_{\$x}.ID = e_{\$x}.ID} full(p_1) $$
$$ \ltimes^n_{e_{\$y}.ID = e_{\$y}.ID} full(p_2) \ltimes^n_{e_{\$y}.ID = e_{\$y}.ID} full(q_2) $$

Finalizing q's translation, we have $alg(q) = xml_{templ}(full(q))$, where xml_{templ} is:

$$ \langle res1 \rangle e_1.C \langle res2 \rangle e_2.C \langle res3 \rangle e_3.C \langle /res3 \rangle \langle /res2 \rangle \langle /res1 \rangle $$

where $e_1.C, e_2.C, e_3.C$ are the C attributes corresponding to the path expressions p_1, p_2 and p_3 (those producing returned nodes). Observe that no data restructuring is needed in xml_{templ}, since the nested joins in $full(q)$ have grouped the data as the query required.

4.3 Isolating patterns from algebraic expressions

Algebraic equivalence rules applied on $(*)$ bring $full(q)$ to the equivalent form:

$$ \sigma_{(e_{\$z}.ID \neq \bot) \vee (e_{\$z}.ID = \bot \wedge e_2 = \bot)}\big(full(p_{\$x}) \ltimes^n_{e_{\$x}.ID \prec e_c.ID} e_c \ \times $$
$$ full(p_{\$y}) \ltimes^n_{e_{\$y}.ID \prec e_e.ID} e_e \ltimes^n_{e_{\$y}.ID \prec e_d.ID} $$
$$ (e_d \ltimes^n_{e_d.ID \prec e_f.ID} (e_f \bowtie^n_{e_f.ID \prec e_g.ID} \sigma_{V=5}(e_g) \ltimes^n_{e_d.ID \prec e_h.ID} e_h))) $$

This rewriting has grouped together $full(p_{\$x})$ with the other subexpressions structurally related to \$x (the join product before the $\times$). It has also grouped $full(p_{\$y})$ and the subexpressions structurally related to \$y (the last two lines). It turns out that these correspond exactly to the algebraic semantics of patterns V_{10} and V_{11} in Figure 1. Thus:

$$ alg(q) = xml_{templ}(\sigma_{(e_{\$z}.ID \neq \bot) \vee (e_{\$z}.ID = \bot \wedge e_2 = \bot)}(V_{10} \times V_{11})) $$

The σ is a by-product of transforming the equality joins in $(*)$ in structural joins.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.
3. S. Amer-Yahia and Y. Kotidis. Web-services architectures for efficient XML data exchange. In *ICDE*, 2004.
4. A. Arion, V. Benzaken, and I. Manolescu. XML Access Modules: Towards Physical Data Independence in XML Databases. XIME-P Workshop, 2005.
5. A. Arion, V. Benzaken, I. Manolescu, and R. Vijay. ULoad: Choosing the Right Store for your XML Application (demo). In *VLDB*, 2005.
6. K. Beyer, F. Ozcan, S. Saiprasad, and B. Van der Linden. DB2/XML: designing for evolution. In *SIGMOD*, 2005.
7. M. Brantner, S. Helmer, C.-C. Kanne, and G.Moerkotte. Full-Fledged Algebraic XPath Processing in Natix. In *ICDE*, 2005.
8. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 2002.
9. Z. Chen, H.V. Jagadish, L. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *VLDB*, 2003.
10. B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, 2001.
11. A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.
12. H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. *VLDB J.*, 11(4), 2002.
13. H. Jiang, H. Lu, W. Wang, and J. Xu. XParent: An efficient RDBMS-based XML database system. In *ICDE*, 2002.
14. R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.
15. I. Manolescu. XML query processing: storage and query model interplay. Tutorial at the EDBT summer school, available at www-rocq.inria.fr/~manolesc, 2004.
16. I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries over heterogeneous data sources. In *VLDB*, 2001.
17. I. Manolescu and Y. Papakonstantinou. An unified tuple-based algebra for XQuery. Available at www-rocq.inria.fr/~manolesc/PAPERS/algebra.pdf, 2005.
18. G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.
19. F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, 2003.
20. P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD*, 2004.
21. S. Paparizos, Y. Wu, L. Lakshmanan, and H. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, 2004.
22. C. Ré, J. Siméon, and M. Fernandez. A complete and efficient algebraic compiler for XQuery. In *ICDE*, 2006.
23. XQuery 1.0. www.w3.org/TR/xquery.
24. W. Xu and M. Ozsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.